**SOCIAPP**

Hamza belyahioui, Fadel Biaou, Abdoul-waris Konate

**Documentation**

Sommaire

**Introduction**

* Résumé du projet
* Diagramme de basse de donnes
* Outils
* Artchitecture cloud
* Development de l’Api
* Perfermanbce et Scalability
* Mise en place de azure
* Point de amélioration
* Conculusion
* Résumé du projet

La démocratisation de l’internet a transformé le monde en village planétaire facilitant ainsi le partage et la communication entre personne proche ou distante. Cependant, cela n’est possible que par des plateformes dédiées sur lesquelles des personnes partagent leur contenu les uns aux autres et plusieurs méthodes de conception et de déploiement existent pour réaliser de telles plateformes.

Pour ce projet, nous avons eu à développer une plateforme de réseau social basé sur le cloud Azure pour une start-up nommé Link Up qui se veut être un système backend évolutif et efficace, qui peut être intégré à n’importe quelle interface frontale, offrant ainsi une expérience de gestion de contenu transparente.

Pour la réalisation de cette plateforme, on a utilisé **Visual Studio** un environnement de developpement sur lequel nous avons implementer notre solution, **GITHUB** pour le versionning et le travail collaboratif , **Draw.io** pour la modelisation de notre solution, du **.Net** pour sa resistantce et aussi son integration rapide avec azure et pour le deploiement nous avons utilisé **Microsoft Azure.**

* Fonctionnalité principale

1. **Authentification et autorisation des utilisateurs** :
   * Gestion des comptes utilisateurs (créateurs).
   * Options de confidentialité pour rendre un profil public ou privé.
2. **Création et gestion de contenu** :
   * Publication, modification, suppression et récupération de contenu.
   * Stockage sécurisé du contenu utilisateur.
3. **Gestion des médias** :
   * Téléchargement et récupération d’images et de vidéos.
4. **Commentaires et modération** :

* Permettre les commentaires sur les publications et leur modération.

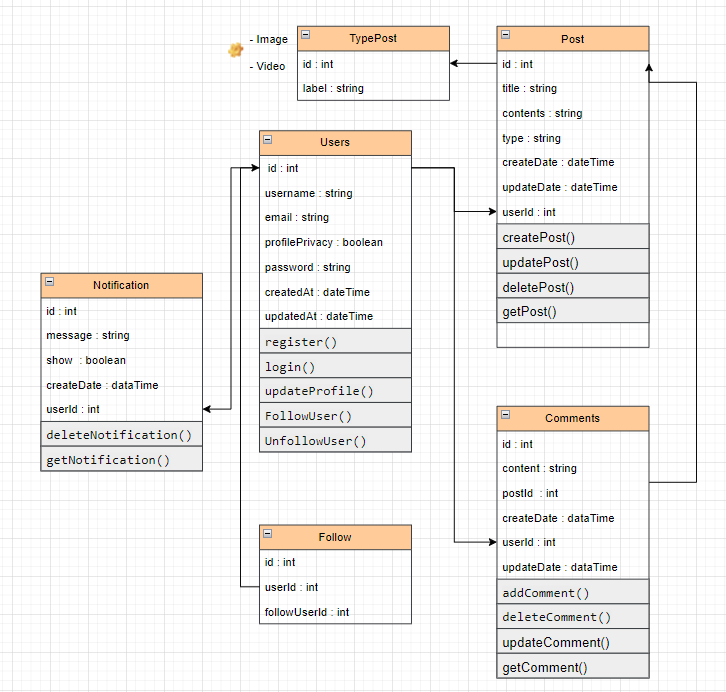
1. **Fonctionnalité de recherche** :

* Rechercher des publications par utilisateur ou par mot-clé.

1. **Notifications** :

Notifier les abonnés des nouvelles publications.

* Diagramme de base de données



**Users** :

Représente les utilisateurs avec leurs informations personnelles (nom, email, etc.) et des actions comme s'inscrire, se connecter, ou suivre d'autres utilisateurs.

**Post** :

Contient les publications créées par les utilisateurs, avec des actions pour créer, mettre à jour ou supprimer des posts.

**Comments** :

Permet aux utilisateurs de commenter les posts, avec des fonctionnalités pour ajouter, supprimer ou modifier un commentaire.

**TypePost** :

Définit le type de publication (image, vidéo, etc.), associé à chaque post.

**Notification** :

Gère les notifications des utilisateurs, comme les messages ou alertes.

**Follow** :

Gère les relations entre les utilisateurs (qui suit qui)

* Outils

1. **Une image contenant Graphique, violet, Caractère coloré, symbole

   Description générée automatiquementVisual Studio**

est un environnement de développement intégré (IDE) de Microsoft, et ASP.NET est un framework de développement web utilisé pour construire des applications web robustes et dynamiques. Ensemble, Visual Studio et ASP.NET offrent un puissant ensemble d'outils pour le développement d'applications web. Voici quelques aspects clés

1. **Github**

Une image contenant logo, Police, Graphique, symbole

Description générée automatiquementPour la gestione de projet on a utilise Github

GitHub est une plateforme web qui utilise Git pour le contrôle de version et la gestion de projets

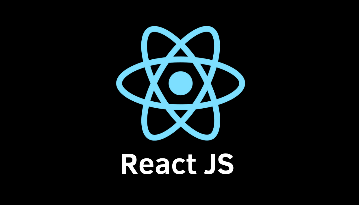
1. **Azure**

**Microsoft Azure** est une plateforme de services cloud complète qui permet de créer, déployer et gérer des applications

1. Une image contenant Bleu électrique, Graphique, bleu, capture d’écran

   Description générée automatiquement**.Net**

Utiliser **.NET avec Azure** est idéal car Azure est parfaitement optimisé pour les applications .NET, offrant une intégration native avec des services comme **Azure App Service, Azure Functions**, et **Azure SQL**, ce qui simplifie le déploiement et la gestion des applications. De plus.

1. **React**

React Js est une bibliothéque Javascript permettant de construire des pages SPA(Single Page Application)

* Une image contenant capture d’écran, diagramme, texte, ligne

  Description générée automatiquementArchitecture Cloud

**User (Utilisateur)**

C'est l'utilisateur final qui interagit avec l'application web via un navigateur ou un appareil.

**Font**

Représente Api web, hébergée sur Azure App Service.

**Api**

Représente Api web, hébergée sur Azure App Service.

**GitHub**

Source du code de l'application.

Avec Azure, vous pouvez configurer une intégration CI/CD (Continuous Integration/Continuous Deployment) via GitHub

**Storage**

Utilisé pour stocker des fichiers, images des videos

**MySQL DB**

Base de données relationnelle utilisée par l'application pour stocker les données, avec Azure Database for MySQL

**Monitor – logs**

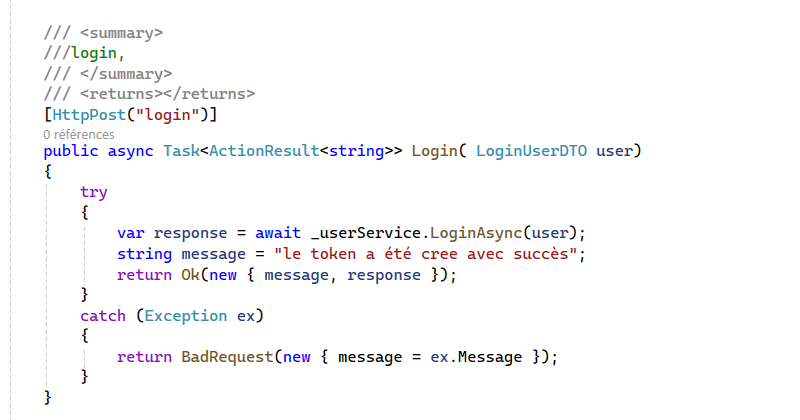
* Development de l’API

Nous avons organisé l’API en deux couches principales :

1. **Controller** : C'est la porte d'entrée où les requêtes arrivent depuis le frontend.
2. **Service** : Ici, j'ai mis toute la logique métier pour éviter d'avoir du code complexe dans le Controller.

Lors de la connexion :

**Controller (Login)** :

* + Le frontend envoie l'email et le mot de passe à l'API.
  + Le Controller appelle la méthode LoginAsync de la couche service et récupère la réponse.
  + Si tout va bien, un token est retourné (JWT) au frontend.
  + Si une erreur se produit, un message d'erreur est envoyé.

**Service (LoginAsync)** :

* + Nous commençons par chercher l’utilisateur dans la base de données via l'email.
  + Si l’utilisateur n’existe pas ou si le mot de passe est incorrect, Nous retournons une erreur.
  + Sinon, un **token JWT** contenant des informations comme l'ID utilisateur et son email est créé.
  + Une image contenant texte, capture d’écran, Police

    Description générée automatiquementEnfin, nous retournons le token.

1. **Créer un Token** :
   * Le token est créé avec une clé secrète définie dans la configuration.
   * Il contient des informations comme l’émetteur, le destinataire, et une date d’expiration



Le token crée est utilisé pour sécuriser les Endpoints :



Nous avons utilisé Authorize pour protéger les routes comme celle qui récupère les posts des utilisateurs. Ça permet de s’assurer que seules les personnes authentifiées, c'est-à-dire celles qui envoient un jeton JWT valide, peuvent y accéder. Quand une requête arrive, ASP.NET vérifie automatiquement le jeton : est-il valide, signé correctement, et non expiré ? Si tout est bon, l’accès est autorisé, sinon la requête est rejetée avec un message d’erreur. En gros, **[Autorise]** garantit que les données sensibles restent sécurisées et accessibles uniquement aux utilisateurs connectés.

Pour enregistrer un utilisateur :

1. Le frontend envoie les informations de l'utilisateur (nom, email, mot de passe, etc.).
2. Dans le Controller, ces données sont validées en appelant la méthode RegisterAsync de la couche service.
3. Une image contenant texte, capture d’écran, Police

   Description générée automatiquementSi tout va bien, l'utilisateur est ajouté à la base de données.

**Étape 4 : Déployer sur Azure avec GitHub Actions**

1. **Déploiement Azure** :
   * Une **App Service** est créée dans Azure pour héberger l’API.
   * La clé secrète JWT, l’émetteur et le destinataire sont ajoutées dans les **paramètres de l’App Service**.
2. **GitHub Actions** :
   * GitHub Actions a été configuré pour automatiser le déploiement.
   * Chaque fois qu’une modification est faite sur la branche principale, GitHub déploie automatiquement l'API sur Azure.

Avec cette API, nous avons :

1. Créer un système d’authentification basé sur JWT pour sécuriser mon API.
2. Permettre l’enregistrement et la connexion d’utilisateurs.
3. Automatiser le déploiement dans Azure via GitHub Actions.

Cela assure que l'API est toujours à jour sans intervention manuelle après chaque mise à jour de code.

* Mise en place du Azure

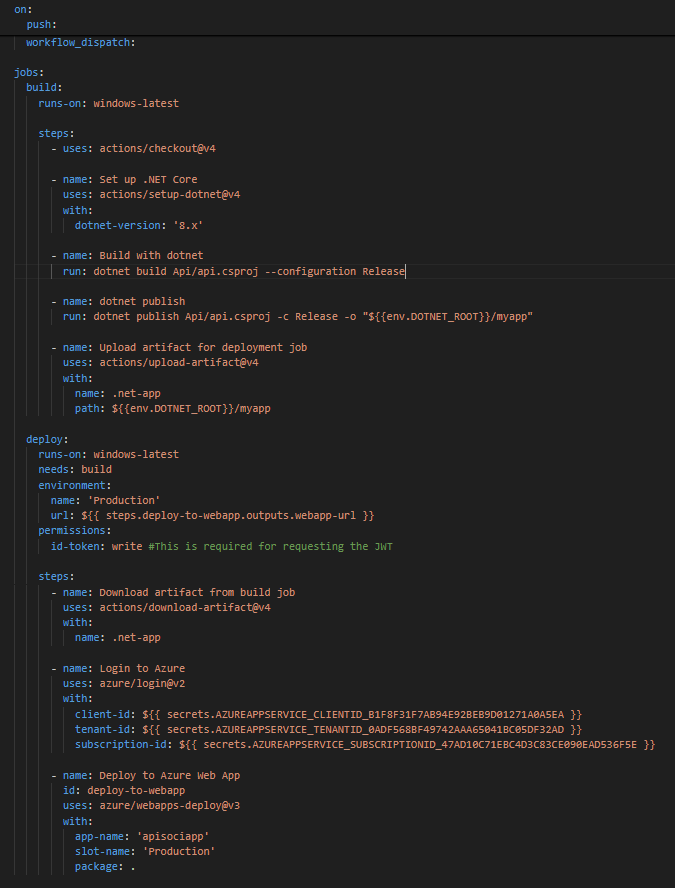
**Api :**

**Création de mon App Service**

**Sur le portail Azure on a créé Un App service**

**Configuration des paramètres nécessaires**

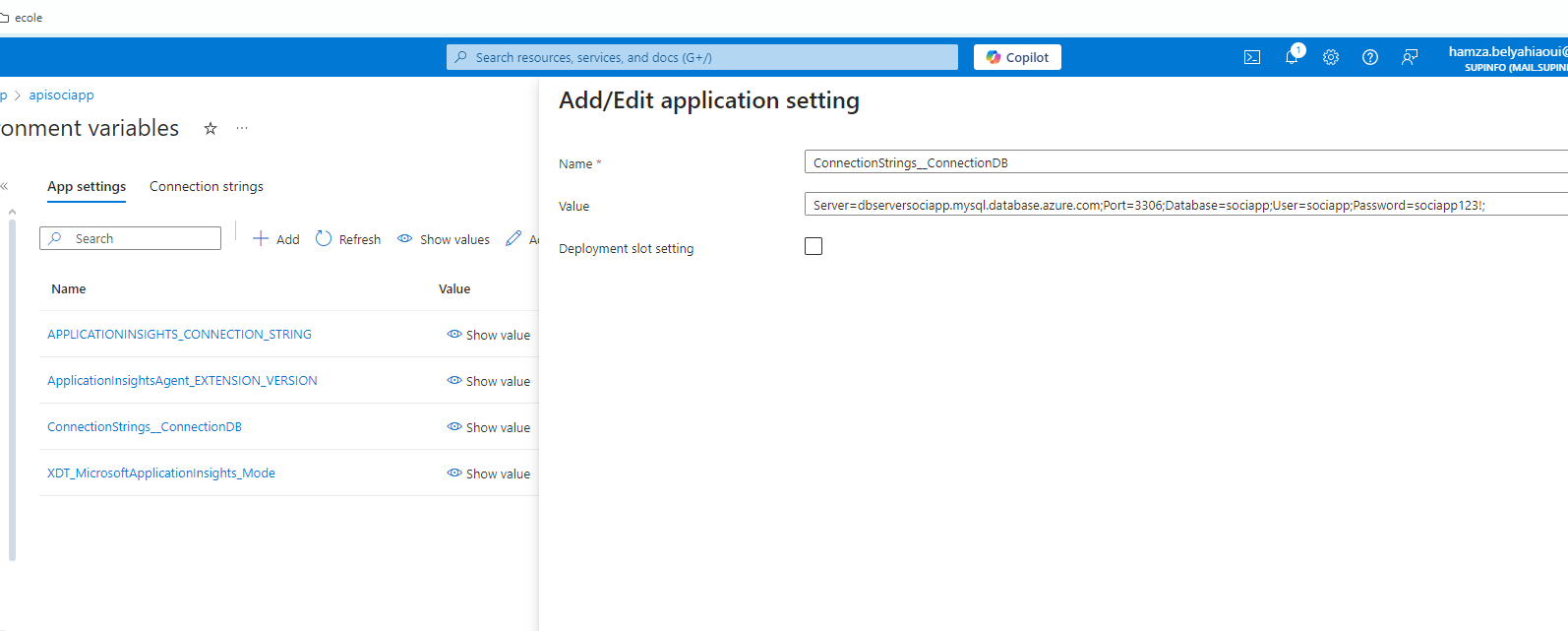
Une fois l'App Service créé, Nous configurons la connexion avec le dépôt GitHub. Cela permet de générer automatiquement un pipeline dans mon workflow. Ce pipeline sera déclenché chaque fois qu’on réalisera un commit sur la branche main.

Ainsi, après chaque commit, le pipeline s'occupera de la construction (build) de l’API, puis procédera au déploiement sur l'App Service. Une fois le déploiement terminé, nous testons mon API en utilisant l'URL fournie par l'App Service.

**DB :**

Nous avons créé une base de données dans Azure en utilisant **Azure Database for MySQL**. Une fois la base de données mise en place, nous la lions à l’application web hébergée dans un App Service sur Azure. Pour ce faire, nous récupérons la chaîne de connexion de la base de données MySQL dans le portail Azure. Cette chaîne contient toutes les informations nécessaires, telles que l'hôte, le nom d'utilisateur, le mot de passe et le nom de la base de données.

Ensuite ajoutons cette chaîne de connexion dans les paramètres de l'App Service de l’application. Nous utilisons, les variables d'environnement pour stocker cette chaîne de connexion afin qu'elle soit disponible au moment de l'exécution de mon API .NET. Cela permet à l'API d'interagir avec la base de données de manière sécurisée et flexible, sans exposer directement les informations sensibles dans le code source.

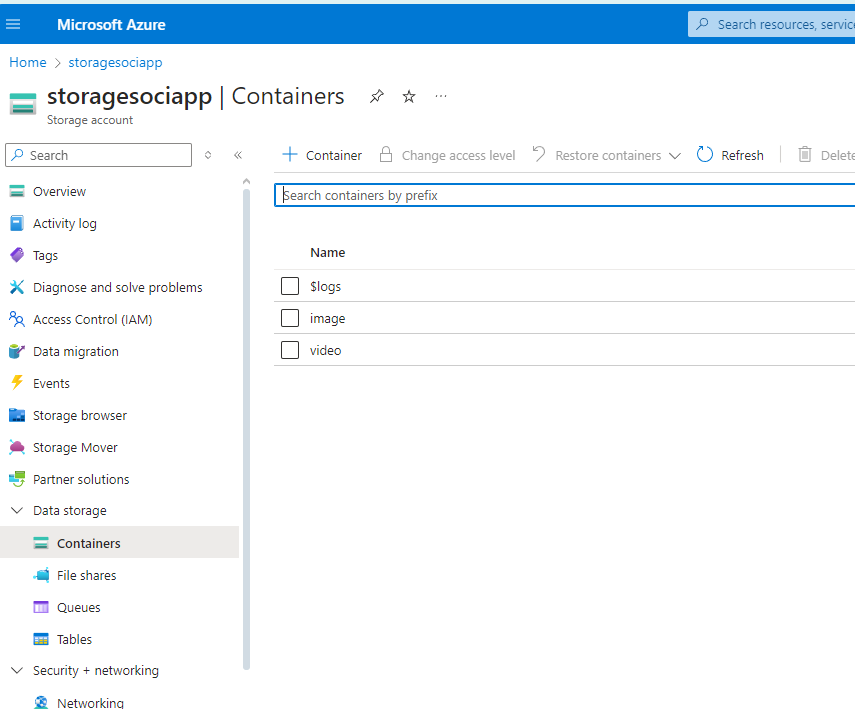


Dans Azure App Service, dans **Setting > Environment variables**.

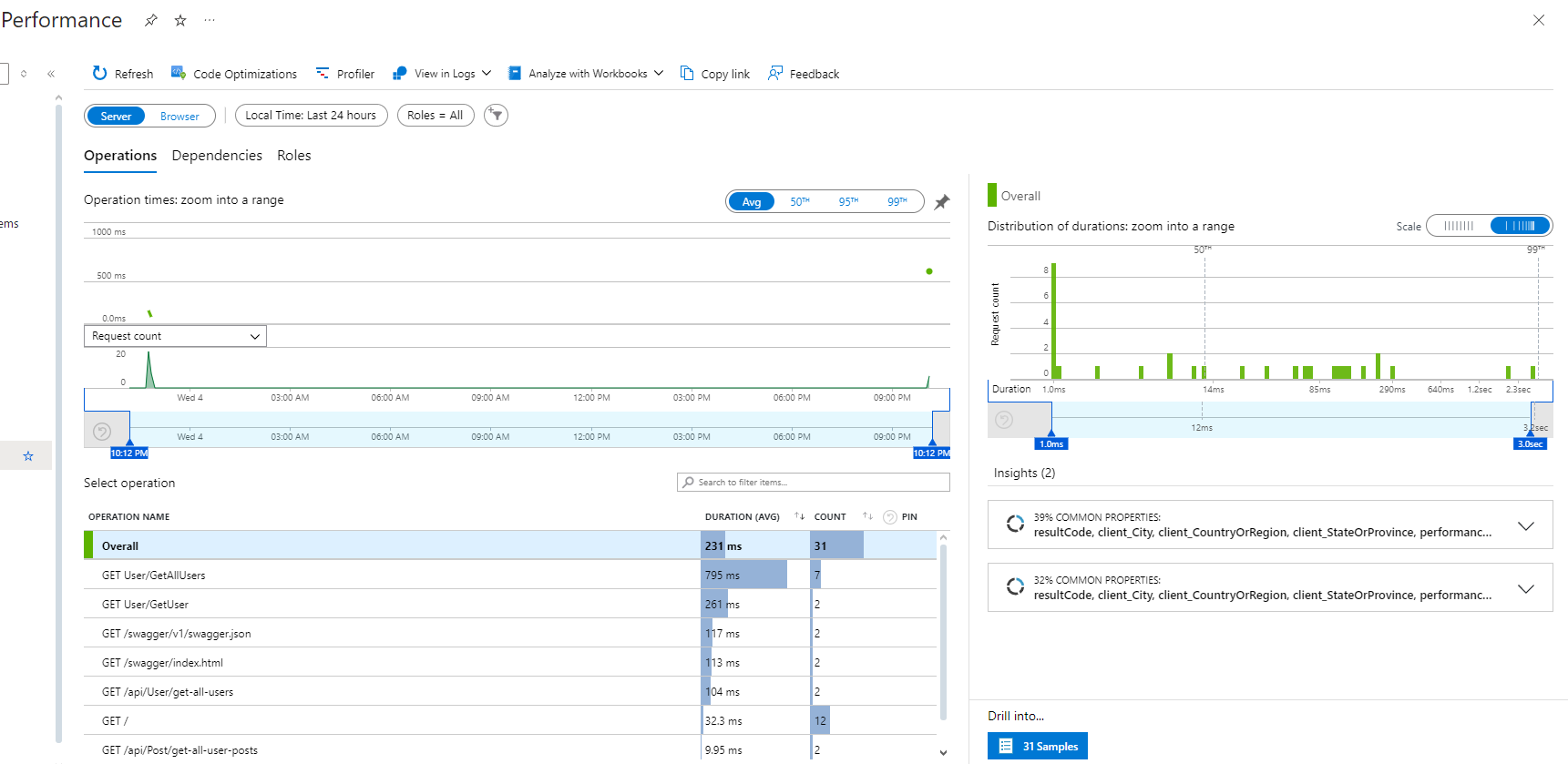
Nous avons ajouter une variable d'environnement **ConnectionString\_\_ConnectionDB** avec la chaîne de connexion.

Storage :

Pour stocker les images et vidéos du projet, nous avons choisi **Azure Blob Storage**. Un **compte de stockage** à d’abord été créé et un **conteneur** sur Azure, puis ajouté la **chaîne de connexion** dans **appsettings.json** du projet en utilisant le package **NuGet**. **Storage.Blobs**, un service pour gérer l'upload des fichiers vers Azure. Ce service stocke les fichiers dans le conteneur et récupère l'URL du fichier, qu’on enregistre ensuite dans la base de données.

Dans l'API, nous ajoutons une méthode pour recevoir les fichiers via **POST** et les stocker dans Azure. Nous avons aussi configuré des **permissions de sécurité** pour contrôler l'accès aux fichiers. Après avoir testé l'upload, l'application a été deployé sur **Azure App Service**, offrant ainsi une solution fiable et sécurisée pour le stockage des fichiers multimédia dans l'application.

* Perfermance et scalability



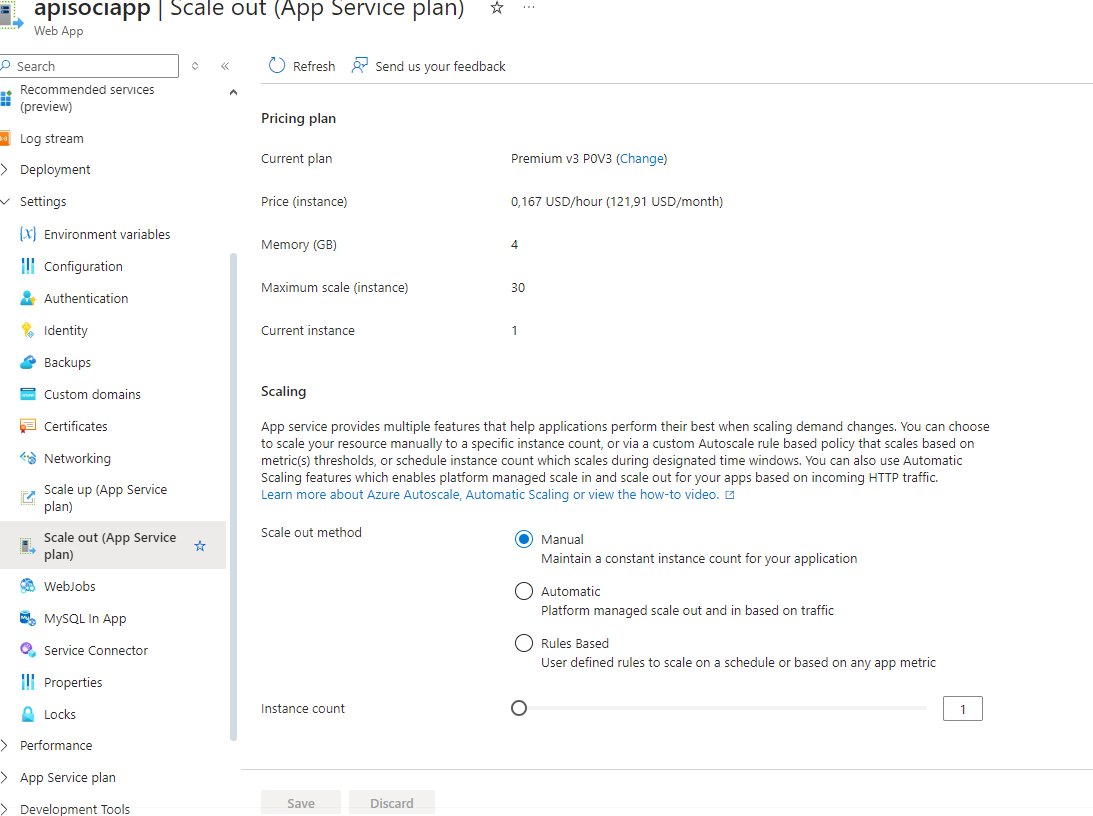
Dans cette analyse de performances via **Azure Monitor**, on constate que l’application, affiche les statistiques des requêtes reçues et traitées.

**Activité récente** : On remarque un pic de requêtes à **22h12**, ce qui peut indiquer une augmentation soudaine du trafic.

**Problèmes de performance** : L’API **GET /User/GetUser** est plus lente, comparée à d’autres endpoints qui répondent rapidement, comme **GET /api/Post/get-all-user-posts**

**Répartition des durées** : La majorité des requêtes sont rapides, mais certaines dépassent **3 secondes**, ce qui pourrait affecter l’expérience utilisateur.

Grâce à ces insights, nous pouvons cibler les endpoints lents pour les optimiser, analyser les logs, et gérer les pics de trafic plus efficacement.

Dans cet écran, la **scalabilité** du service Web Azure App Service a été configuré. Actuellement sur un **plan tarifaire Premium v3 P0V3**, qui coûte **0,167 USD par heure**, soit environ **121,91 USD par mois**. Ce plan nous donne une mémoire allouée de **4 Go** et permet d’avoir jusqu’à **30 instances** simultanées au maximum.

Pour la configuration de la mise à l'échelle, trois options principales s’offrent à nous :

1. **Manuelle** : Avec cette option, nous pouvons maintenir un nombre constant d’instances pour l’application. Cela permet d’ajuster manuellement les ressources utilisées en fonction des besoins. Actuellement, j’ai défini **1 instance** active.
2. **Automatique** : Cette option permettrait à Azure de gérer la scalabilité de manière autonome. Azure ajouterait ou supprimerait des instances en fonction du trafic et des besoins de l’application.
3. **Basée sur des règles (Rules Based)** : Avec cette méthode, je pourrais définir mes propres règles de mise à l'échelle basées sur des paramètres comme l’utilisation du CPU, la mémoire, ou des plages horaires spécifiques. Cela me permettrait d'optimiser les performances tout en contrôlant les coûts.

Pour l’instant, nous avons choisi de rester en mode **manuel**, mais nous gardons la possibilité de passer à l’un des modes dynamiques si nos besoins évoluent, notamment si le trafic augmente de façon significative."

* Points améliorations

Le projet pourrait se voir amélioré en 3 grande phase :

* **Plan Fonctionnel**

Ajouter une map pour connaitre la position, Intégrer un chat ce qui facilitera les échanges entre les créateurs et ceux qui les suivent, mettre à jour les **feeds**, Permettre les likes

* **Plan architectural**

Ajouter **Azure Key Vault** pour mieux sécuriser les informations sensibles, en optimisant la gestion des logs avec **Application Insights**, équilibrer les charges à l’aide de **loadBalancer** ou **Api Gateway.** Aussi, on peut conteneuriser les différentes parties de la solution et les déployer à l’aide de **Azure kubernetes Services,** et **Azure Cognitive Services** pour rajouter un service Ia de détection d’images par exemple. Et pour finir, renforcer l’intégration avec **GITHUB ACTIONS** par des workflows (Tests unitaires et d'intégration, Analyse de sécurité avec des outils comme Dependabot, Déploiements progressifs (Blue-Green ou Canary) pour éviter les interruptions de service)et utiliser **Ansible ou Terraform**

* **Plan managérial**

Pour le monitoring, Visualisation via **Power BI** peut être ajoutée, avoir une idée de l’état de santé avec **Azure Health,** Azure Blob Storage tiers de performance pour catégoriser les fichiers **(Hot, cool, cold, archive)** et **Alertes proactives**

Conclusion

Ce projet social, réalisé avec une API en .NET et un frontend en ReactJs, a été déployé avec les solutions Azure, telles qu’App Service pour l'API, **Azure Storage** pour les données non structurées et **Azure Database for MySQL** pour la gestion des données. Le processus de déploiement a été assuré via **GitHub Actions**.